

Lehrstuhl für Informatik 10 (Systemsimulation)



**The Math Library of the *pe* Physics Engine –
Combining Smart Expression Templates and BLAS
Efficiency**

Klaus Iglberger, Ulrich Rüde

The Math Library of the *pe* physics engine – Combining Smart Expression Templates and BLAS Efficiency

Klaus Iglberger, Ulrich Rüde

Lehrstuhl für Systemsimulation
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

klaus.iglberger@informatik.uni-erlangen.de

December 7, 2009

Performance optimized linear algebra operations are an essential ingredient for all numerical packages. Therefore several standard frameworks are focused on the performance optimized implementation of these operations, as for instance several packages implementing the BLAS standard. However, next to performance, also reliability, stability, usability, and maintainability are important aspects of such a framework, which in many cases only play a subordinate role in the development.

In this report we introduce the smart expression template programming technique used in the math library of the *pe* rigid multibody physics engine. This technique offers an expression-dependent evaluation and optimization of mathematical vector and matrix operations, the automatic detection of aliasing effects, and combines the usability and maintainability of standard C++ programming with the efficiency of an optimized BLAS library. We will demonstrate the efficiency of this approach by comparing the *pe* math library with the Boost uBLAS library.

1 Motivation

A lot of CPU time in numerical simulations is spent in the execution of basic linear algebra subprograms (BLAS), such as vector additions, matrix-vector multiplications, and matrix-matrix multiplications. Obviously, the performance optimized implementation of these operations is crucial for efficient simulations. For this purpose the BLAS standard was introduced to define a set of the most important linear algebra operations. BLAS is focused on providing an interface for highly optimized linear algebra functionality. Since performance is platform dependent, the BLAS standard has been implemented in various libraries, as for instance ATLAS [2], Intel's IMKL [10], AMD's ACML [1], or NVidia's CUBLAS [13]. All of these frameworks implement the set of BLAS operations defined by the BLAS standard according to its interface. However, all of them are primarily focused on performance, yet neglect several other important aspects of software frameworks, such as usability and maintainability. The following code example shows the signature of the C interface function for the multiplication of two dense, double precision matrices according to the expression $C = \alpha \cdot A \cdot B + \beta \cdot C$:

Listing 1: Signature of the BLAS function for the matrix-matrix multiplication for double precision matrices

```
1 void cblas_dgemm( const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
2                 const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
3                 const int K, const double alpha, const double *A,
4                 const int lda, const double *B, const int ldb,
5                 const double beta, double *C, const int ldc );
```

The trouble with the BLAS functions is the lack of usability and maintainability. The generality of the function formulation due to the language deficiencies of C and the goal to enable all kinds of matrix-matrix multiplications makes it harder to use this function and still harder for maintainers to understand the performed operation. Usability is also limited due to the general restriction to the two data types `float` and `double`¹ and the restriction to dense data structures. Integral data types and mixed types, as for instance for the multiplication between a `float` and a `double` matrix and sparse data structures are not supported at all. Additionally, it is up to the programmer to make sure the sizes of the matrices match, i.e. the function has no possibility to check for possible errors.

Listing 2: Usage of the BLAS matrix-matrix multiplication

```
1 // Definition of the matrices A, B, and C
2 double* A = ...;
3 double* B = ...;
4 double* C = ...;
5
6 // Initialization of the matrices
7 ...
8
9 // Performing the multiplication
10 // Are the parameters correct? Are the matrix sizes valid?
11 cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans, 3, 2, 3, 1.0, A, 3, B, 2, 0, C, 2 );
```

A different approach is taken by the Boost framework’s uBLAS library [4, 3]. This library is not primarily focused on high performance but on providing user-friendly and maintainable BLAS level 1, 2, and 3 functionality for dense, packed, and sparse vectors and matrices. Their C++ design and implementation unifies mathematical notation via operator overloading and efficient code generation via expression templates.

Listing 3: Matrix-matrix multiplication using the Boost uBLAS library

```
1 using boost::numeric::ublas;
2
3 // Definition of the matrices A, B, and C
4 matrix<double> A, B, C;
5
6 // Initialization of the matrices
7 ...
8
9 // Performing the multiplication
10 C = A * B;
```

In contrast to the multiplication via the `cblas_dgemm` function, the intent of the programmer is perfectly clear in this case: the matrices `A` and `B` are multiplied and the result is written to matrix `C`. The complexity of the operation, including all performance optimizations, are hidden behind this intuitive interface. Another usability advantage in comparison to the BLAS standard is that even if the matrices are changed to single precision matrices (`matrix<float>`), the multiplication itself remains unchanged. This is even true for mixed type matrix-matrix multiplications:

Listing 4: Mixed type matrix-matrix multiplication using the Boost uBLAS library

```
1 using boost::numeric::ublas;
2
3 // Definition of the matrices A, B, and C
4 matrix<double> A, C;
5 matrix<float> B;
6
7 // Initialization of the matrices
8 ...
9
10 // Performing the mixed type multiplication
11 C = A * B;
```

¹In case you also consider complex numbers as data type, single and double precision complex numbers are also supported.

In addition, even if the sizes of the matrices are changed in the initialization part, the overloaded multiplication operator makes sure that only matrices of matching size are multiplied. This gives an important security factor that cannot be provided by the BLAS functions. However, even for the default BLAS functionality for dense vectors and matrices, the performance of the Boost uBLAS implementation severely lacks behind the other BLAS implementations.

In this report, we introduce the math library of the *pe* rigid multibody physics engine [11]. This library combines the usability and maintainability of a C++ code as implemented in the Boost library, and the performance of an optimized BLAS implementation. This is achieved by a smart expression template based implementation that allows expression dependent optimizations. In the following Section 2, the basic idea of expression templates is explained in details, followed by an in-depth analysis of smart expression templates in Section 3. Section 4 shows performance results for the *pe* math library in comparison to the Boost uBLAS library and the ATLAS library [2]. Section 5 concludes the report.

2 Expression Templates

Expression templates are a C++ template programming technique to improve the performance degrading characteristics of the standard operator overloading technique. The expression template technique was concurrently invented by Todd Veldhuizen and David Vandevoorde in 1995 [14, 15, 16]. The basic idea of expression templates is the delay of the computation of overloaded operators by creating intermediate expression objects that represent the result of a numerical operation. Only if these intermediate objects are assigned to their destination, the evaluation of the expression is performed. This approach circumvents the creation of a temporary object, which usually consists of a memory allocation, a copy operation, and a memory deallocation. Since these expression objects can be efficiently inlined by the compiler, expression template based C++ code can attain 95-99.5% efficiency in comparison to hand-crafted C code and speed improvements of 2-15 times in comparison to conventional C++ code [8].

To illustrate the technique of expression templates, the following example will first recall the classical approach of operator overloading, before demonstrating the addition of dense vectors via expression objects.

Listing 5: Basic implementation of a vector class

```

1 template< typename Type > // Type of the vector elements
2 class Vector
3 {
4     public:
5         explicit Vector( std::size_t n, Type value=Type() )
6             : n_( n )
7             , v_( new Type[n] )
8         {
9             std::fill( v_, v_+n, value );
10        }
11
12        Vector( const Vector& v )
13            : n_( v.n_ )
14            , v_( new Type[n_] )
15        {
16            std::copy( v.v_, v.v_+n_, v_ );
17        }
18
19        ~Vector() {
20            delete[] v_;
21        }
22
23        Vector& operator=( const Vector& rhs )
24        {
25            if( &rhs == this ) return *this;
26
27            resize( rhs.size_ );
28            std::copy( rhs.v_, rhs.v_+n_, v_ );
29
30            return *this;
31        }
32
33        std::size_t size() const { return n_; }

```

```

34     Type&      operator[]( size_t i )      { return v_[i]; }
35     const Type& operator[]( size_t i ) const { return v_[i]; }
36
37     void resize( std::size_t n ) { /*...*/ }
38
39     // ...
40
41 private:
42     std::size_t n_; // The current size/dimension of the vector
43     Type*      v_;  // The dynamically allocated vector elements
44 };

```

Listing 5 shows the relevant part of the basic implementation of a classical **Vector** class for elements of arbitrary type. A **Vector** can be either created by explicitly specifying its size **n** or via copy construction. Additionally, the class provides a copy assignment operator, functions for the direct access to the vector elements, and a function to change the size of a vector.

Listing 6: Addition operator for the addition of two vectors

```

1 template< typename Type >
2 const Vector<Type> operator+( const Vector<Type>& lhs, const Vector<Type>& rhs )
3 {
4     if( lhs.size() != rhs.size() )
5         throw std::invalid_argument( "Vector sizes do not match" );
6
7     Vector<Type> tmp( lhs.size() );
8     for( std::size_t i=0; i<lhs.size(); ++i )
9         tmp[i] = lhs[i] + rhs[i];
10
11     return tmp;
12 }

```

Listing 6 shows the according addition operator for the addition of two **Vectors**². The first step is a check whether the sizes of the two vectors match; if not, a `std::invalid_argument` exception is thrown. Otherwise, a temporary vector **tmp** is created and filled with the sum of the two involved vectors, before it is returned.

It is this temporary value that is made responsible for the performance penalty of the C++ implementation in comparison to a C or Fortran implementation [12]. The creation of the temporary value results in a dynamic memory allocation and deallocation, and in a vector copy operation during the initialization of the function return value by the return expression which is accomplished with copy initialization [5]. However, note that this is only half of the story since the temporary value doesn't have to be created in case the return value is used to create another vector. In this case, the “named return value optimization” (NRV) as described in section 12.1.1c of the ARM [12], is used to optimize the code:

Listing 7: Application of NRV during copy construction and copy assignment

```

1 Vector<double> a, b;
2 Vector<double> c( a + b ); // No temporary value necessary due to NRV
3
4 c = a + b; // Creation of a temporary value

```

If a compiler applies the NRV to a code (which is triggered by the presence of an explicit copy constructor), the local variable **tmp** will be replaced by a reference to the eventual destination of the return value in the caller. It is as if the function were written as follows:

Listing 8: Standard implementation of the vector addition operator

```

1 template< typename Type >
2 const Vector<Type>
3     operator+( Vector<Type>& dest, const Vector<Type>& lhs, const Vector<Type>& rhs )
4 {
5     // ...
6
7     dest.Vector<Type>::Vector( lhs.size() ); // Explicit call of the constructor
8
9

```

²Note that this particular definition of the addition operator is only able to add two vectors of the same element type. By using the `MathTrait` class template (see Appendix B) it would be easily possible to add two vectors of different element types.

```

10     for( std::size_t i=0; i<lhs.size(); ++i )
11         tmp[i] = lhs[i] + rhs[i];
12 }

```

In Listing 7, during the creation of `c`, the compiler is able to directly copy the result of the addition into vector `c`. However, this is not possible in the case of the assignment, since the copy assignment operator is a member function that performs operations similar to a destruction of `c` followed by a reinitialization of it. Optimizing the assignment could therefore severely alter the semantics of the program (i.e. remove side effects of the copy assignment operator). Therefore the compiler is forced to create a temporary, which results in code similar to the following one:

Listing 9: Compiler generated code for the copy assignment of vectors

```

1 Vector<double> a, b, c;
2 Vector<double> tmp( a + b ); // NRV optimized addition of a and b into the temporary tmp
3 c = tmp;                    // Assignment of the temporary to the vector c

```

The performance penalty from the generation of temporaries even increases if several additions are concatenated:

Listing 10: Concatenation of multiple vector additions

```

1 Vector<double> a, b, c, d;
2
3 d = a + b + c;

```

In this case a temporary vector is created for every single expression. Therefore the overhead of the vector addition grows considerably: one memory allocation and deallocation, one copy operation, and one loop execution for every addition expression. This overhead is not avoidable for the classical C++ formulation of the vector addition, although the final result could be calculated in a single `for`-loop (as it is usually done in a plain C implementation):

Listing 11: C-like implementation of multiple vector additions

```

1 Vector<double> a, b, c, d;
2
3 for( size_t i=0; i<size; ++i ) {
4     d[i] = a[i] + b[i] + c[i];
5 }

```

The approach of expression templates is to remove the creation of the temporary object entirely and to delay the execution of the expression until it is assigned to its target. Therefore the addition operator no longer returns the result of the addition, but a temporary object that acts as a placeholder for the addition expression [8]:

Listing 12: Expression template formulation of the vector addition

```

1 template< typename A, typename B >
2 class Sum
3 {
4     public:
5         explicit Sum( const A& a, const B& b )
6             : a_( a )
7             , b_( b )
8         {}
9
10        std::size_t size() const { return a_.size(); }
11        double operator[]( std::size_t i ) const { return a_[i] + b_[i]; }
12
13    private:
14        const A& a_; // Reference to the left-hand side operand
15        const B& b_; // Reference to the right-hand side operand
16 };
17
18 template< typename A, typename B >
19 Sum<A,B> operator+( const A& a, const B& b )
20 {
21     if( a.size() != b.size() )
22         throw std::invalid_argument( "Vector sizes do not match" );
23
24     return Sum<A,B>( a, b );
25 }

```

Instead of calculating the result of the addition of two vectors, the addition operator now returns an object of type `Sum<A,B>`, where `A` and `B` are the types of the left- and right-hand side operands. The only requirements the addition operator poses on `A` and `B` are the existence of a subscript operator and a `size` function. The `Sum` class now temporarily represents the addition, until an assignment operator is encountered:

Listing 13: Expression template assignment operator

```

1 template< typename Type > // Type of the vector elements
2 class Vector
3 {
4     public:
5         // ...
6
7         template< typename A >
8         Vector& operator=( const A& expr )
9         {
10             resize( expr.size() );
11             for( std::size_t i=0; i<expr.size(); ++i )
12                 v_[i] = expr[i];
13
14             return *this;
15         }
16
17         // ...
18 };

```

This assignment operator is the only other assignment operator of the `Vector` class next to the copy assignment operator (which is strictly necessary due to the management of the dynamically allocated memory). Every time an expression object ³ is assigned to a `Vector`, the assignment operator from Listing 13 is used to handle the assignment. It first resizes the vector accordingly and afterwards traverses the elements of the given expression within a single `for`-loop. Note that this traversal triggers the evaluation of the expression due to the accesses of the values via the subscript operator. Also note that this `for`-loop is the only `for`-loop necessary to evaluate the entire expression.

Via this formulation based on the inline formulation of all functions and the evaluation within a single `for`-loop hidden in the assignment operator it is possible to attain the performance of a C-like implementation as illustrated in Listing 11. It is even possible to concatenate several additions without the creation of any temporary (and still a single `for`-loop evaluation):

Listing 14: Expression template optimized addition of multiple vectors

```

1 Vector<double> a, b, c, d;
2
3 c = a + b + c; // Evaluated in a single for-loop

```

The expression instantiation graph that is created during the compilation of the right-hand side expression `a + b + c` is illustrated in Fig. 1. For the addition of the first two vectors, a temporary object of type `Sum< Vector<double>, Vector<double> >` is instantiated, that represents the addition of the two vectors `a` and `b`. For the addition with the third vector `c`, a temporary expression object of type `Sum< Sum< Vector<double>, Vector<double> >, Vector<double> >` is created, the represents the addition of all three vectors.

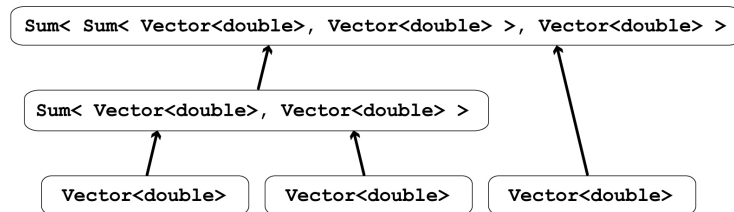


Figure 1: Expression template instantiation graph for the addition of three vectors.

³The thorough reader might notice that due to the signature of this assignment operator all non-vector objects assigned to a vector that do not fit the signature of the copy assignment operator will use this assignment operator. The problems associated with this fact will be handled in the remainder of this section.

Note that the expression objects returned instead of the immediate result of the operation are temporary objects as well. However, their creation and the according copy operations don't hurt the performance for two reasons. The first reason is the fact that these temporary expression objects don't contain any member data that are expensive to copy. Instead they only contain references to the operands of the according numerical operation. The second reason is the aforementioned NRV. Since the expression objects are not assigned but instead created, the compiler has leeway to optimize the copy operation away entirely. Listing 15 demonstrates this for the addition of three vectors as shown in Listing 14.

Listing 15: Compiler generated temporaries during the addition of three vectors

```

1 Vector<double> a, b, c, d;
2
3 // Compiler generated temporary for the expression object returned by the addition of
4 // vector a and b. Due to NRV no copy operation is required!
5 const Sum< Vector<double>, Vector<double> > __tmp1__( a + b );
6
7 // Compiler generated temporary for the expression object returned by the addition of
8 // the first addition of a and b and the third vector c. Again, due to NRV no copy
9 // operation is required!
10 const Sum< Sum< Vector<double>, Vector<double> >, Vector<double> > __tmp2__( __tmp1__ + c );
11
12 // Assignment of the right-hand side expression to the left-hand side vector d
13 d = __tmp2__;

```

In terms of the vector addition, the shown expression template formulation is already complete and working. Figure 2 shows a performance comparison between the classical operation overloading technique and the demonstrated expression template formulation for the addition of three vectors as performed in Listing 14⁴.



Figure 2: Performance comparison between the classical operation overloading technique and expression templates for the addition of three vectors. The performance was measured on an Intel Core i7 940 CPU at 2.93 Ghz (Bloomfield core) with 8 MByte of shared level three cache using double precision vector elements. The executables were compiled with the GNU G++ 4.4.1 compiler (branch revision 150839).

However, due to the general formulation of the addition operator and the assignment operator of the `Vector` class, this addition operator now also qualifies for additions between completely unrelated types:

Listing 16: Imperfection of the general expression template formulation

```

1 Vector<double> x, y;
2 Matrix<double> A;
3
4 y = A + x;

```

Although it should not be possible to add a matrix and a vector, the current formulation of the addition operator allows this operation. In order to prevent the arbitrary addition of unrelated types, an `Expr` base class is introduced [8]:

Listing 17: Final expression template formulation of the vector addition

```

1
2 template< typename A >
3 struct Expr
4 {
5     const A& operator~() const {
6         return *static_cast<const A*>( this );
7     }
8 };
9
10

```

⁴The test machine for this performance measurement was an Intel Core2 Quad CPU (Q6700) with 2.66GHz.


```

11 template< typename Type > // Type of the vector elements
12 class Vector : public Expr< Vector<Type> >
13 {
14     public:
15         // ...
16
17     template< typename A >
18     Vector& operator=( const Expr<A>& expr )
19     {
20         const A& a( ~expr );
21
22         resize( a.size() );
23         for( std::size_t i=0; i<a.size(); ++i )
24             v_[i] = a[i];
25
26         return *this;
27     }
28
29     // ...
30 };
31
32 template< typename A, typename B >
33 class Sum : public Expr< Sum<A,B> >
34 {
35     public:
36     explicit Sum( const A& a, const B& b )
37         : a_( a )
38         , b_( b )
39     {}
40
41     std::size_t size() const { return a_.size(); }
42     double operator[]( std::size_t i ) const { return a_[i] + b_[i]; }
43
44     private:
45     const A& a_; // Reference to the left-hand side operand
46     const B& b_; // Reference to the right-hand side operand
47 };
48
49 template< typename A, typename B >
50 Sum<A,B> operator+( const Expr<A>& a, const Expr<B>& b )
51 {
52     if( (~a).size() != (~b).size() )
53         throw std::invalid_argument( "Vector sizes do not match" );
54
55     return Sum<A,B>( ~a, ~b );
56 }

```

Both the `Vector` and the `Sum` class now publicly derive from the `Expr` base class and are therefore considered valid operands for the addition operator. The `Expr` base class is based on the “Curiously Recurring Template Pattern” (CRTP) [16], which enables a type-safe downcast to the dynamic type of the expression object. In the implementation of the *pe* this downcast is explicitly performed via the `operator~`, as for example used in the addition operator. Note that in this improved expression template formulation, both the addition operator as well as the assignment operator are now only accepting `Expr` objects and therefore effectively restrict the number of possible arguments to the allowed expressions.

3 Smart Expression Templates

Smart expression templates are an extension of the expression template formulation shown in Section 2. The term “smart” in this sense refers to the fact that smart expression templates (in contrast to standard expression templates) can optimize the expression evaluation depending on the type of the operands (see 3.1). Additionally, the smart expression templates as implemented in *pe* are able to detect aliasing effects and adapt the expression evaluation accordingly (Section 3.2). Furthermore, smart expression templates allow an integration of performance optimized BLAS functionality in order to combine the usability and maintainability of the expression templates with the high performance of BLAS (see Section 3.3).

3.1 Expression Dependent Evaluation

The expression template formulation as shown in the previous chapter works fine for simple expressions, as for instance the addition of vectors, the multiplication between a matrix and a vector, or a matrix-matrix multiplication. However, for complex or composite expressions the standard formulation can lead to severe performance penalties. One example is the following matrix-vector multiplication:

Listing 18: Complex matrix-vector multiplication

```
1 Matrix<double> A;  
2 Vector<double> a, b, c;  
3  
4 c = A * ( a + b );
```

Using the expression template formulation as shown before, this right-hand side expression would result in an expression object that contains references to its two operands: the matrix **A** and the vector addition **a + b**. However, for the evaluation of this expression, the entire result of the vector addition is needed for every single element of the **c** vector. Therefore each element of the vector addition is evaluated **M** times, where **M** is the number of rows of the matrix. This unfortunately decreases the performance of the expression template formulation considerably. Figure 3 shows the performance results of the composite matrix-vector multiplication for the expression in Listing 18. Obviously, the standard formulation of expression templates is slower than the classical approach that involves two vector temporaries (one for the vector addition, one for the matrix-vector multiplication), although during the expression template evaluation no memory allocations/deallocations or expensive copy operations take place. Included in the figure are also the performance result for the smart expression template implementation as explained in this section and the result for the combination of smart expression templates with a performance optimized BLAS implementation.

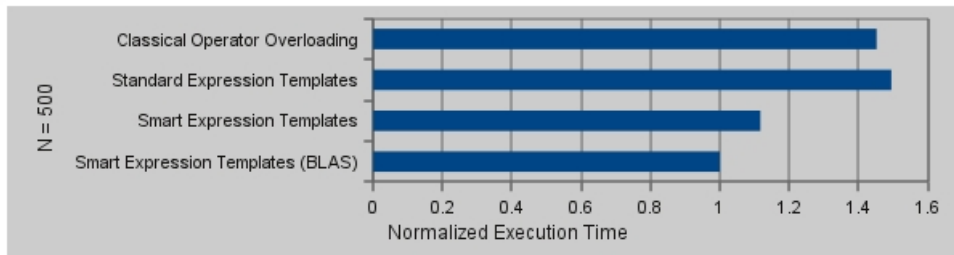


Figure 3: Performance comparison between the several implementations of a dense matrix-vector multiplication. The performance was measured on an Intel Core i7 949 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache and using double precision elements for both the matrix and the vectors. All executables were compiled with the GNU G++ 4.4.1 compiler (branch revision 150839).

Apparently it is necessary to distinguish between a matrix-vector multiplication between a matrix and a vector and a matrix-vector multiplication between a matrix and a vector expression. Therefore the expression classes are updated by a mechanism to detect whether or not the operands are expressions or plain objects. Listing 19 shows part of the smart expression template implementation of the matrix-vector addition as implemented in the *pe*:

Listing 19: Smart expression object for the matrix-vector multiplication

```
1 template< typename MT      // Type of the left-hand side dense matrix  
2           , typename VT > // Type of the right-hand side dense vector  
3 class DMatDVecMultExpr : public DenseVector< DMatDVecMultExpr<MT,VT> >  
4                       , private Expression  
5 {  
6     private:  
7         // Result type of the left-hand side dense matrix expression  
8         typedef typename MT::ResultType      MRT;  
9  
10        // Result type of the right-hand side dense vector expression  
11        typedef typename VT::ResultType       VRT;  
12  
13        // Element type of the left-hand side dense matrix expression  
14        typedef typename MRT::ElementType     MET;  
15  
16
```

```

17 // Element type of the right-hand side dense vector expression
18 typedef typename VRT::ElementType VET;
19
20 // ...
21
22 public:
23 // Result type for expression template evaluations.
24 typedef typename MathTrait<MRT,VRT>::MultType ResultType;
25
26 // Data type for composite expression templates
27 typedef const DMatDVecMultExpr& CompositeType;
28
29 // Resulting element type
30 typedef typename ResultType::ElementType ElementType;
31
32 // Member data type of the left-hand side dense matrix expression
33 typedef typename MT::CompositeType Lhs;
34
35 // Member data type of the right-hand side dense vector expression
36 typedef typename SelectType<IsExpression<VT>::value,const VRT,const VT&>::Type Rhs;
37
38 inline const ElementType operator[]( size_t index ) const;
39
40 inline size_t size() const;
41
42 // ...
43
44 private:
45 Lhs mat_; // Left-hand side dense matrix of the multiplication expression
46 Rhs vec_; // Right-hand side dense vector of the multiplication expression
47
48 // ...
49 };

```

The `DMatDVecMultExpr` class template represents the multiplication of a dense matrix and a dense vector (either plain matrices/vectors or matrix/vector expressions). This class is publicly derived from the `DenseVector` class template that works in the same way as the `Expr` base class of the standard expression template formulation and allows a restriction of operands to dense vectors. Additionally, the `DMatDVecMultExpr` class derives privately from the `Expression` base class. The purpose of this base class is solely to mark the `DMatDVecMultExpr` class as a vector expression in comparison to a plain vector.

The major difference in comparison to standard expression templates is an additional set of nested type definitions that are required for a smart evaluation of expressions. The `MRT` and `VRT` defined in a private section of the class are merely shortcuts for the result types of the left-hand side and right-hand side operands defined by the nested `ResultType` type definition. The `ResultType` represents the resulting data type in case the expression object is evaluated. Note that these types are never expression types but always concrete types (vectors, matrices, ...) that have to be supported by the `MathTrait` class template (see Appendix B). The result type of an expression is defined as the data type resulting from the according operation between the left-hand side operand of type `RT1` and the right-hand side operand of type `RT2`. The nested type `ElementType` represents the element type of the expression, that is derived from the `ResultType` of the expression. `MET` and `VET` are shortcuts to the element type of the left-hand side and right-hand side result types.

The nested `CompositeType` type represents the data type that is used in case the expression is used in another expression (as for instance another addition, subtraction, etc.). This data type is either a reference to the expression itself in case it is not necessary to evaluate the expression in case of a composite expression, or it is an according concrete type (in most cases the `ResultType`) in case it is necessary or beneficial to immediately evaluate the expression within a composite expression. In case of the `DMatDVecMultExpr` class, the `CompositeType` is a reference to `DMatDVecMultExpr` since it is not necessary to evaluate a matrix-vector multiplication prior to any composite evaluation and is not beneficial in terms of performance. However, in case of an addition between a dense and a sparse vector, the `CompositeType` is a concrete dense vector type, since it is a huge performance advantage to evaluate the expression immediately:

Listing 20: Composite type of the smart expression object for the dense vector-sparse vector addition

```

1 template< typename VT1      // Type of the left-hand side dense vector
2       , typename VT2 >    // Type of the right-hand side sparse vector
3 class DVecSVecAddExpr : public DenseVector< DVecSVecAddExpr<VT1,VT2> >
4       , private Expression
5 {
6     public:
7         // ...
8
9         // Result type for expression template evaluations.
10        typedef typename MathTrait<RT1,RT2>::AddType   ResultType;
11
12        // Data type for composite expression templates.
13        typedef const ResultType                       CompositeType;
14
15        // ...
16 };

```

In the `DMatDVecMultExpr` class, the `CompositeType` is for instance used to determine the member data type of the left-hand side operand (`Lhs`). In case an expression needs to be immediately evaluated, `Lhs` becomes a temporary that holds the result of the left-hand side expression. In case the expression does not have to be evaluated immediately, it corresponds to a reference to the left-hand side matrix operand.

The left-hand side dense matrix operand does not require any special treatment. In the matrix-vector multiplication every element of the matrix is used exactly once. Therefore the decision whether or not to evaluate the matrix expression is handled by the nested `CompositeType`. The right-hand side dense vector expression on the other hand requires a special treatment. Depending on whether or not the right-hand side vector is a plain vector or a vector expression, the right-hand side member `Rhs` has to be chosen differently: in case the right-hand side vector operand is a plain vector, it is enough to store a reference to the vector inside the matrix-vector multiplication object. However, if the right-hand side vector is an expression type (i.e. derived from the `Expression` base class) a temporary is created to hold the result of the vector expression in order to guarantee an efficient evaluation of the matrix-vector multiplication.

The type definition of the member data type for the right-hand side dense vector operand `Rhs` demonstrates how the type dependent evaluation of expressions works. The `Rhs` type is defined as either the result type of the vector expression (`VRT`) or a reference to the right-hand side plain vector via the `SelectType` class template (see Appendix A). The selection between these two types is performed by the `IsExpression` type trait:

Listing 21: Type trait class for expressions

```

1 template< typename T >
2 struct IsExpressionHelper
3 {
4     enum { value = boost::is_base_of<Expression,T>::value &&
5                 !boost::is_base_of<T,Expression>::value };
6     typedef typename SelectType<value,TrueType,FalseType>::Type   Type;
7 };
8
9 template< typename T >
10 struct IsExpression : public IsExpressionHelper<T>::Type
11 {
12     public:
13         enum { value = IsExpressionHelper<T>::value };
14         typedef typename IsExpressionHelper<T>::Type   Type;
15 };

```

The `IsExpression` type trait class tests whether or not the given type `Type` is a *pe* expression template or not. In order to qualify as a valid expression template, the given type has to derive (publicly or privately) from the `Expression` base class. This relationship is tested via the Boost `is_base_of` type trait. In case the given type is a valid expression template, the `value` member enumeration is set to 1, the nested type definition `Type` is `TrueType`, and the class derives from `TrueType`. Otherwise `value` is set to 0, `Type` is `FalseType`, and the class derives from `FalseType`.

For the evaluation of the right-hand side member type of the matrix-vector multiplication object, the `SelectType<IsExpression<VT>::value,const VRT,const VT&>` results either in `const VRT` in case the right-hand side vector `VT` is an expression or in `const VT&` in case the right-hand side vector is a plain

vector.

An obvious difference in comparison to standard expression templates is the creation of temporary objects as members of the expression objects. At first sight this might be a potentially expensive endeavor since the expression objects are returned by value (i.e. by a copy operation) from the overloaded operators. However, also in this case, the NRV optimization removes any potential copy operation and directly constructs the object into a compiler generated, temporary value:

Listing 22: Compiler generated temporaries for the matrix-vector multiplication

```
1 Matrix<double> A;
2 Vector<double> a, b, c;
3
4 // Compiler generated temporary for the expression object returned by the addition of
5 // vector a and b. No copy operation is required due to NRV!
6 const DVecDVecAddExpr< Vector<double>, Vector<double> > __tmp1__( a + b );
7
8 // Compiler generated temporary for the expression object returned by the multiplication
9 // of a dense matrix with a dense vector expression. Due to NRV, no copy operation is
10 // required!
11 const DMatDVecMultExpr< Matrix<double>, Vector<double> > __tmp2__( A * __tmp1__ );
12
13 // Assignment of the right-hand side expression to the left-hand side vector c
14 c = __tmp2__;
```

3.2 Detection of Aliasing Effects

The standard expression template formulation has a second weakness that is directly correlated to the correctness of the calculations. Consider the multiplication of a dense matrix and a dense vector as illustrated in Listing 23:

Listing 23: Aliasing problem for a matrix-vector multiplication

```
1 Matrix<double> A;
2 Vector<double> x;
3
4 x = A * x; // Aliasing effect!!
```

In the standard formulation, this calculation would most certainly result in a wrong result. During the evaluation of the right-hand side multiplication expression, the values of the left-hand side vector are directly updated without the creation of an intermediate, temporary result. Therefore the values of `x` are changed during the expression evaluation although the old values of `x` are still required. In the standard expression template formulation this problem can only be solved by an explicit generation of a temporary, since it is not possible to detect such aliasing problems:

Listing 24: Explicit resolution of aliasing problems

```
1 Matrix<double> A;
2 Vector<double> x;
3
4 Vector<double> tmp( A * x ); // Explicit creation of a temporary vector, optimized by NRV!
5 x = tmp; // Update of vector x!
```

However, it is potentially very dangerous to allow the formulation of incorrect calculations.⁵ Therefore it is necessary to incorporate the detection of aliasing effects into the expression template evaluation. In the *pe* implementation, this is solved by an additional member function in every vector, matrix, and expression class, called `isAliased`:

Listing 25: Member function to detect aliasing effects

```
1 template< typename MT // Type of the left-hand side dense matrix
2           , typename VT > // Type of the right-hand side dense vector
3 class DMatDVecMultExpr : public DenseVector< DMatDVecMultExpr<MT,VT> >
4                       , private Expression
5 {
6     // ...
```

⁵Even a perfect documentation of this issue could not prevent any accidental use of this formulation even by experienced programmers.

```

7
8 public:
9     // ...
10
11     template< typename T >
12     inline bool isAliased( const T* alias ) const {
13         return vec_.isAliased( alias );
14     }
15
16     // ...
17
18 private:
19     Lhs mat_; // Left-hand side dense matrix of the multiplication expression
20     Rhs vec_; // Right-hand side dense vector of the multiplication expression
21
22     // ...
23 };

```

The `isAliased` function of the `DMatDVecMultExpr` class returns whether any operand of the expression is aliased with the given address `alias`. In case of the matrix-vector multiplication, aliasing effects can only occur in case the expression is assigned to a vector that is part of the multiplication expression. Therefore the function relays the query to the vector (expression). Listing 26 illustrates the implementation of the `isAliased` function in the `Vector` class:

Listing 26: Implementation of the `isAliased` function in the `Vector` class

```

1 template< typename Type > // Data type of the vector
2 template< typename Other > // Data type of the foreign expression
3 inline bool Vector<Type>::isAliased( const Other* alias ) const
4 {
5     return static_cast<const void*>( this ) == static_cast<const void*>( alias );
6 }

```

The initial call to the `isAliased` function is contained in the assignment operator of the vector and matrix classes. For instance, consider the according assignment operator of the `Vector` class:

Listing 27: Smart expression template assignment operator of the `Vector` class

```

1 template< typename VT > // Type of the right-hand side dense vector
2 inline Vector& Vector::operator=( const DenseVector<VT>& rhs )
3 {
4     using pe::assign;
5
6     if( IsExpression<VT>::value && (~rhs).isAliased( this ) ) {
7         Vector tmp( rhs );
8         swap( tmp );
9     }
10    else {
11        resize( (~rhs).size(), false );
12        assign( *this, ~rhs );
13    }
14
15    return *this;
16 }

```

This smart expression assignment operator represents the direct replacement of the standard expression template assignment operator demonstrated in Listing 17. It exclusively accepts dense vectors and dense vector expressions, i.e. classes deriving from the `DenseVector` class. The assignment operation distinguishes between a default assignment and an aliased assignment. In case the given right-hand side dense vector is an expression, an alias test is performed with the address of the left-hand side vector. In case this test returns `true`, it is necessary to create a temporary and afterwards update the values of the vector via the “temporary swap” idiom [7, 6]. If no aliasing is detected, the vector is resized accordingly and the right-hand side is directly assigned without any temporary (the `assign` functionality is discussed in the next section).

Note that self-assignment between two vectors ($x = x$) is handled by the copy assignment operator. Therefore the `IsExpression<VT>::value` expression should be considered a performance optimization since it is not possible to encounter any aliasing effects in case the right-hand side dense vector is not an expression. Also note that this implementation of the detection of aliasing effects is based on the assumption that the addresses of the vectors and matrices are uniquely identifying a specific vector or

matrix. In the current implementation this is particularly true since both base classes of the **Vector** class are empty base classes and therefore don't occupy any memory on their own in case the compiler supports the "Empty Base Class Optimization" [17]. However, this property is assured by several compile time constraints [9].

3.3 Integration of BLAS Functionality

In order to achieve a maximum level of performance, the smart expression template formulation has to allow the expression dependent customization of the assignment. This would for instance enable the integration of BLAS functionality to specific operations. For the purpose of this section, let's consider the multiplication between two dense matrices since this particular operation profits the most from the use of a performance optimized BLAS library. Listing 28 shows the assignment operator of the **Matrix** class that is very similar to the assignment operator of the **Vector** class (Listing 27):

Listing 28: Smart expression template assignment operator of the **Matrix** class

```

1 template< typename Type > // Type of the matrix elements
2 template< typename MT > // Type of the right-hand side dense matrix
3 inline Matrix<Type>& Matrix<Type>::operator=( const DenseMatrix<MT>& rhs )
4 {
5     using pe::assign;
6
7     if( IsExpression<MT>::value && (~rhs).isAliased( this ) ) {
8         MatrixMxN tmp( rhs );
9         swap( tmp );
10    }
11    else {
12        resize( (~rhs).rows(), (~rhs).columns() );
13        assign( *this, ~rhs );
14    }
15
16    return *this;
17 }
```

Depending on the result of the **if** statement that handles the detection of aliasing effects, either a temporary is created and afterwards "assigned" via the "temporary swap" idiom [7, 6], or the matrix is resized and afterwards assigned the right-hand side dense matrix (expression). The **assign** function is the key to the expression dependent optimization of the assignment. **assign** is a free function expecting as arguments the left-hand side target matrix and the right-hand side dense matrix (expression) to be assigned to the left-hand side matrix. The default implementation of **assign** is illustrated in Listing 29:

Listing 29: Default implementation of the free **assign** function

```

1 template< typename MT1 // Type of the left-hand side dense matrix
2           , typename MT2 > // Type of the right-hand side dense matrix
3 inline void assign( DenseMatrix<MT1>& lhs, const DenseMatrix<MT2>& rhs )
4 {
5     pe_INTERNAL_ASSERT( (~lhs).rows() == (~rhs).rows() , "Invalid_number_of_rows" );
6     pe_INTERNAL_ASSERT( (~lhs).columns() == (~rhs).columns(), "Invalid_number_of_columns" );
7
8     (~lhs).assign( rhs );
9 }
```

The **assign** function expects the target matrix to have the appropriate size for the assignment. Therefore the **resize** operation is performed before the assignment in the assignment operator. Afterwards, the default assignment strategy for a dense matrix is selected, which is depending on the implementation of the left-hand side dense matrix and therefore a member function of the left-hand side matrix. The strategy for the assignment is therefore a ping-pong approach: the assignment operator calls a free function, that allows an expression dependent customization (for instance via overloading or specialization) and in the default case the default assignment is selected, which is again a member function of the **Matrix** class:

Listing 30: Default implementation of the assignment of a dense matrix

```

1 template< typename Type > // Type of the matrix elements
2 template< typename MT > // Type of the right-hand side dense matrix
3 inline void Matrix<Type>::assign( const DenseMatrix<MT>& rhs )
4 {
5     const size_t end( (~rhs).columns() & size_t(-2) );
6 }
```

```

7   for( size_t i=0; i<(~rhs).rows(); ++i ) {
8       for( size_t j=0; j<end; j+=2 ) {
9           v_[i*n_+j ] = (~rhs)(i,j );
10          v_[i*n_+j+1] = (~rhs)(i,j+1);
11      }
12      if( end < (~rhs).columns() ) {
13          v_[i*n_+end] = (~rhs)(i,end);
14      }
15  }
16 }

```

The default assignment applies an element-wise strategy to assign the new values. However, this strategy is a bad choice for the evaluation of a matrix-matrix multiplication. Therefore it will be necessary to customize this strategy explicitly. Note that even this function allows a certain level of optimization by applying explicit loop unrolling.

In order to optimize the assignment of a matrix-matrix multiplication, the expression class has to provide an according `assign` function. The following two code excerpts explain in detail how this optimization is implemented in case of the `DMatDMatMultExpr` class that represents the multiplication between two dense matrices:

Listing 31: Smart expression object for the matrix-matrix multiplication

```

1  template< typename MT1    // Type of the left-hand side dense matrix
2          , typename MT2 > // Type of the right-hand side dense matrix
3  class DMatDMatMultExpr : public DenseMatrix< DMatDMatMultExpr<MT1,MT2> >
4          , private Expression
5  {
6  private:
7      // Result type of the left-hand side dense matrix expression
8      typedef typename MT1::ResultType      RT1;
9
10     // Result type of the right-hand side dense matrix expression
11     typedef typename MT2::ResultType      RT2;
12
13     // Composite type of the left-hand side dense matrix expression
14     typedef typename MT1::CompositeType   CT1;
15
16     // Composite type of the right-hand side dense matrix expression
17     typedef typename MT2::CompositeType   CT2;
18
19  public:
20     // Result type for expression template evaluations
21     typedef typename MathTrait<RT1,RT2>::MultType   ResultType;
22
23     // Data type for composite expression templates
24     typedef const DMatDMatMultExpr&                CompositeType;
25
26     // Resulting element type
27     typedef typename ResultType::ElementType       ElementType;
28
29     // Member data type of the left-hand side dense matrix expression.
30     typedef typename SelectType<IsExpression<MT1>::value,const RT1,CT1>::Type   Lhs;
31
32     // Member data type of the right-hand side dense matrix expression.
33     typedef typename SelectType<IsExpression<MT2>::value,const RT2,CT2>::Type   Rhs;
34
35     // ...
36
37     inline const ElementType operator()( size_t i, size_t j ) const;
38     inline size_t rows() const;
39     inline size_t columns() const;
40     inline bool isAliased( const T* alias ) const;
41
42     // ...
43
44  private:
45     Lhs lhs_; // Left-hand side dense matrix of the multiplication expression.
46     Rhs rhs_; // Right-hand side dense matrix of the multiplication expression.
47
48     // ...

```


Just as any other expression class of the *pe* the `DMatDMatMultExpr` class defines the nested types `ResultType`, `CompositeType`, and `ElementType`. Additionally, it defines the two member data types for the left-hand and right-hand side operands `Lhs` and `Rhs` depending on the given types. This process is similar to the type evaluation of the `DMatDVecMultExpr` class, since also for a matrix-matrix multiplication the creation of temporary values in case either of the two operands is an expression increases the performance. As a dense matrix, the `DMatDMatMultExpr` class also provides a function call operator for the access to the matrix elements, a `rows` and a `columns` function, and the `isAliased` function to detect aliasing effects.

Listing 32 shows the customized optimization of the `assign` function. For the purpose of providing a customized `assign`, the `DMatDMatMultExpr` class defines four nested friend functions. At the point of instantiation of the class, these function are injected into the surrounding namespace and are considered by the compiler during the selection of matching candidates for `assign` function calls (this approach is also known as the Barton-Nackman trick [16]):

Listing 32: Assign functions for the evaluation of the matrix-matrix multiplication

```

1  // ...
2
3  template< typename MT > // Type of the target dense matrix
4  friend inline void assign( DenseMatrix<MT>& lhs, const DMatDMatMultExpr& rhs )
5  {
6      pe_INTERNAL_ASSERT( (~lhs).rows()      == rhs.rows()      , "Invalid_number_of_rows" );
7      pe_INTERNAL_ASSERT( (~lhs).columns()    == rhs.columns()    , "Invalid_number_of_columns" );
8
9      if( rhs.lhs_.columns() == 0 ) {
10         reset( ~lhs );
11         return;
12     }
13
14     DMatDMatMultExpr::assign( ~lhs, rhs.lhs_, rhs.rhs_ );
15 }
16
17 template< typename MT3      // Type of the left-hand side target matrix
18         , typename MT4      // Type of the left-hand side matrix operand
19         , typename MT5 >    // Type of the right-hand side matrix operand
20 static inline void assign( MT3& C, const MT4& A, const MT5& B )
21 {
22     // Default implementation of the matrix-matrix implementation
23     // ...
24 }
25
26 template< template<typename> class MT3      // Type of the left-hand side target matrix
27         , template<typename> class MT4      // Type of the left-hand side matrix operand
28         , template<typename> class MT5 >    // Type of the right-hand side matrix operand
29 static inline void assign( MT3<float>& C, const MT4<float>& A, const MT5<float>& B )
30 {
31     const size_t M( A.rows() );
32     const size_t N( B.columns() );
33     const size_t K( A.columns() );
34     cblas_sgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, 1.0F,
35                 &A(0,0), K, &B(0,0), N, 0.0F, &C(0,0), N );
36 }
37
38 template< template<typename> class MT3      // Type of the left-hand side target matrix
39         , template<typename> class MT4      // Type of the left-hand side matrix operand
40         , template<typename> class MT5 >    // Type of the right-hand side matrix operand
41 static inline void assign( MT3<double>& C, const MT4<double>& A, const MT5<double>& B )
42 {
43     const size_t M( A.rows() );
44     const size_t N( B.columns() );
45     const size_t K( A.columns() );
46     cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, 1.0,
47                 &A(0,0), K, &B(0,0), N, 0.0, &C(0,0), N );
48 }
49
50 // ...
51 };
```

The first of the four `assign` functions is a template for an expression dependent optimization of the assignment of a specific expression. The function is an overload to the default `assign` function. In

contrast to the default function, the second argument is not a template argument, but an instance of a `DMatDMatMultExpr`. In case a matrix-matrix multiplication is assigned to another dense matrix, this function is selected to handle the assignment to the right-hand side matrix instead of the default function. In case of the `DMatDMatMultExpr`, this function acts as a dispatcher. It merely calls one of the three provided `assign` member functions of the `DMatDMatMultExpr` class. These functions are overloaded such that in case the two operands of the multiplication match the arguments expected from one of the two BLAS matrix-matrix multiplication functions `cblas_sgemm` or `cblas_dgemm`, the according member function is called and the BLAS functions are executed. In case the BLAS functions cannot be used, the default performance optimized `assign` function is called. In contrast to the default `assign`, which employs an element-wise strategy for the assignment, this function implements a cache optimized calculation of the matrix values.

Note that the same strategy can also be used for addition assignments and subtraction assignments:

Listing 33: Addition and subtraction assignment of a matrix-matrix multiplication

```

1 Matrix<double> A, B, C, D;
2
3 C += A * B;
4 D -= A * B;
```

The according functions are called `addAssign` and `subAssign`, respectively, and follow the same implementation approach as the `assign` function family.

4 Performance Results

In this section, the performance of the smart expression templates as implemented in the math library of the *pe* are examined in more detail. The focus of this section is the performance of composite expressions. Appendix C gives a complete overview of possible operations between dense and sparse vectors and matrices. The *pe* performance results are compared to the performance of the uBLAS library of the Boost framework. This library also offers an expression template based implementation of several dense and sparse vector and matrix data structures. However, so far Boost is primarily focused on usability and maintainability, and on providing a rich set of features for all kinds of mathematical operations. In contradiction to the name of the library, it does not yet support any high performance BLAS operations ⁶.

Listing 34 shows a code excerpt for the performance benchmark of the complex expression $(A + B) \cdot (C - D)$ using the uBLAS library. Prior to the performance measurement, all matrices are accordingly sized and initialized. Afterwards, a certain number of repetitions are executed, each time measuring the passed wall clock time with the timing functionality of the *pe*. Each operation is executed several times to guarantee runtimes of several seconds per measurement. Additionally, special care is taken to prevent the compiler to optimize the calculations away.

Listing 34: Performance benchmark for the dense matrix-dense matrix multiplication using Boost uBLAS

```

1 boost::numeric::ublas::matrix<double> A( N, N ), B( N, N ), C( N, N ), D( N, N ), E( N, N );
2 pe::timing::WcTimer timer;
3
4 // Initialization of the matrices
5 // ...
6
7 // Initial calculation of the matrix E
8 noalias( E ) = prod( A + B, C - D );
9
10 for( std::size_t rep=0; rep<reps; ++rep ) {
11     timer.start();
12     for( std::size_t step=0; step<steps; ++step ) {
13         noalias( E ) = prod( A + B, C - D );
14     }
15     timer.end();
16 }
```

The test machine for all performance tests was an Intel Core i7 940 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache. All executables are compiled with the GNU G++ 4.4.1 compiler

⁶Although this might change with any new release of the Boost library.

(branch revision 150839). The data types of the vector and matrix elements is `double` for all performance tests. We used Boost 1.39 for all performance comparisons and the used BLAS library is ATLAS in the version 3.9.17.

4.1 Complex Expression $A \cdot (a + b)$

Listing 35: Implementation of the expression $A \cdot (a + b)$ in the *pe*

```

1 // Definition of a double precision dense square matrices
2 // and three double precision dense vectors
3 pe::MatN A( N, N );
4 pe::VecN a( N ), b( N ), c( N );
5
6 // Initialization
7 // ...
8
9 // Evaluation of the expression
10 c = A * ( a + b );

```

The complex expression $A \cdot (a + b)$ as already used in the motivation part benefits from the smart expression template formulation due to the intermediate evaluation of the vector expression $a + b$ into a temporary object. In comparison to the uBLAS library, the smart expression template formulation of the *pe* in combination with the BLAS `cblas_dgemv` function for double precision matrix-vector multiplications performs better by a factor of 1.41 for small matrices and vectors ($N = 50$) and by a factor of 1.47 for large matrices and vectors ($N = 1000$). The major performance improvement in this case results from the creation of the temporary. The additional application of a performance optimized BLAS library only slightly improves the performance.

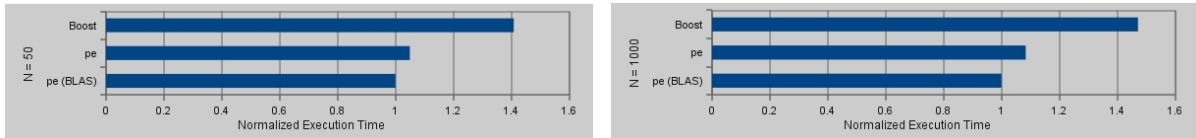


Figure 4: Performance comparison between the *pe* and Boost for the complex expression $A \cdot (a + b)$.

4.2 Complex Expression $A \cdot (a + b + c)$

Listing 36: Implementation of the expression $A \cdot (a + b + c)$ in the *pe*

```

1 // Definition of a double precision dense square matrices
2 // and four double precision dense vectors
3 pe::MatN A( N, N );
4 pe::VecN a( N ), b( N ), c( N ), d( N );
5
6 // Initialization
7 // ...
8
9 // Evaluation of the expression
10 d = A * ( a + b + c );

```

For the multiplication of a matrix with a vector expression representing the addition of three vectors, the performance gain of the smart expression template formulation is even more prominent. In comparison to the uBLAS library, the *pe* performs better by a factor of 1.84 for small operands ($N = 50$) and by a factor of 1.88 for large operands ($N = 1000$). Again, the application of a performance optimized BLAS library only slightly improves the performance in comparison to the performance gain from the smart expression template formulation.

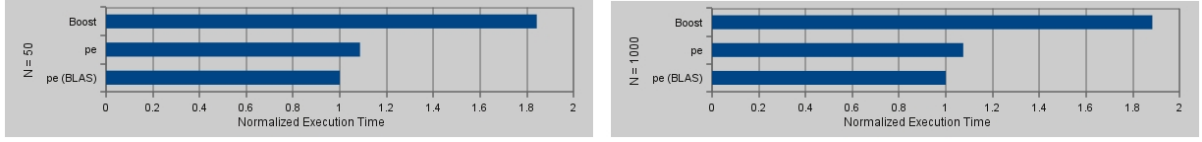


Figure 5: Performance comparison between the *pe* and Boost for the complex expression $A \cdot (a + b + c)$.

4.3 Complex Expression $(A \cdot B) \cdot (a + b)$

Listing 37: Implementation of the expression $(A \cdot B) \cdot (a + b)$ in the *pe*

```

1 // Definition of two double precision dense square matrices
2 // and three double precision dense vectors
3 pe::MatN A( N, N ), B( N, N );
4 pe::VecN a( N ), b( N ), c( N );
5
6 // Initialization
7 // ...
8
9 // Evaluation of the expression
10 c = ( A * B ) * ( a + b );

```

The third complex expression represents a matrix-vector multiplication between a dense matrix and a dense vector, where the dense matrix is the result of a matrix-matrix multiplication and the dense vector results from the addition of two dense vectors. Although even without usage of the optimized BLAS functions the *pe* is faster than the BOOST uBLAS library by a factor of 1.15 for small operands ($N = 50$) and by a factor of 1.11 for larger operands ($N = 400$), the main performance improvement results from the ATLAS library. Note that although each element of the matrix resulting from the matrix-matrix multiplication is required only once during the subsequent matrix-vector multiplication, the creation of a temporary matrix for the result in order to be able to apply the BLAS functionality is very beneficial for the overall performance of the expression. Also note that the performance gain is higher the larger the operands of the expressions are.

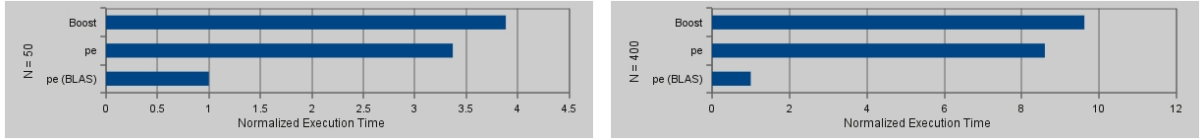


Figure 6: Performance comparison between the *pe* and Boost for the complex expression $(A \cdot B) \cdot (a + b)$.

4.4 Complex Expression $(A \cdot B) + C$

Listing 38: Implementation of the expression $(A \cdot B) + C$ in the *pe*

```

1 // Definition of four double precision dense square matrices
2 pe::MatN A( N, N ), B( N, N ), C( N, N ), D( N, N );
3
4 // Initialization
5 // ...
6
7 // Evaluation of the expression
8 D = ( A * B ) + C;

```

Also the fourth complex expression $(A \cdot B) + C$ greatly profits from the performance optimized BLAS functions. Whereas without the BLAS functions, the *pe* and uBLAS exhibit a similar performance, the application of ATLAS greatly improves the performance, especially for large operands. Also in this case, the performance gain is only possible by creating a temporary matrix for the result of the matrix-matrix multiplication in order to be able to apply the BLAS functionality.

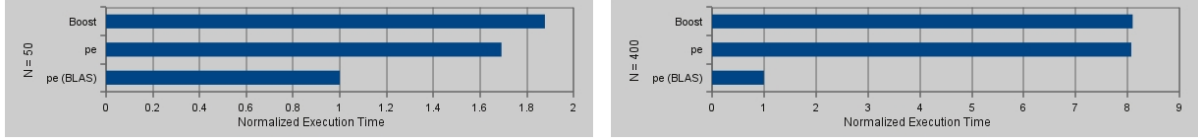


Figure 7: Performance comparison between the *pe* and Boost for the complex expression $(A \cdot B) + C$.

4.5 Complex Expression $(A + B) \cdot (C - D)$

Listing 39: Implementation of the expression $(A + B) \cdot (C - D)$ in the *pe*

```

1 // Definition of five double precision dense square matrices
2 pe::MatN A( N, N ), B( N, N ), C( N, N ), D( N, N ), E( N, N );
3
4 // Initialization
5 // ...
6
7 // Evaluation of the expression
8 E = ( A + B ) * ( C - D );

```

The complex expression $(A + B) \cdot (C - D)$ impressively demonstrates the advantage of smart expression templates for the evaluation of composite expressions. The matrix addition $A + B$ is multiplied with the result of the matrix subtraction $C - D$. Since for the matrix-matrix multiplication all elements from both matrices are required several times, both operands are evaluated into temporary matrices. Only this approach already results in a performance difference of a factor of 1.48 for small matrices ($N = 50$) and a factor of 14.36 for large matrices ($N = 500$) in favor of the *pe* in comparison to Boost uBLAS. In case ATLAS is used, another factor of 3.52 for small matrices and 4.78 for large matrices is gained. In total, this results in a performance difference of a factor of 68.62 between the *pe* and uBLAS.

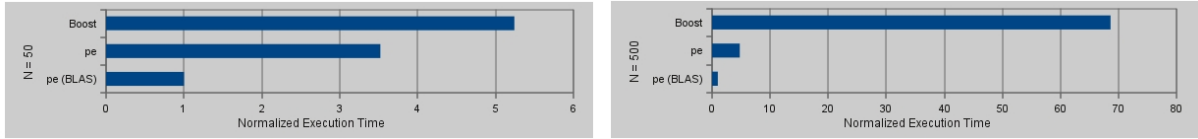


Figure 8: Performance comparison between the *pe* and Boost for the complex expression $(A + B) \cdot (C - D)$.

5 Conclusion

In this report we introduced the smart expression template programming technique used in the math library of the *pe* rigid multibody physics engine. In comparison to standard expression templates as they are used to circumvent the performance degrading characteristics of the C++ technique of operator overloading they offer an expression dependent evaluation and optimization including the possible integration of BLAS functionality, and the automatic detection of aliasing effects. We presented an in depth explanation of the implementation details of smart expression templates and gave an overview of the extensions necessary to update standard expression templates. In order to demonstrate the achieved performance of this approach, we compared the performance of several complex expressions between the *pe* math library and the Boost uBLAS library. In comparison to uBLAS, which offers similar features as the *pe* math library and is also based on expression templates, smart expression templates in combination with BLAS may result in tremendous performance improvements depending on the given expression.

A The SelectType class template

The `SelectType` class template is a template metaprogramming class to select one of two given types depending on a boolean compile time constant expression. The implementation is based on a class template and a single specialization:

Listing 40: Implementation of the SelectType class

```

1 template< bool Select    // Compile time selection
2     , typename T1      // Type to be selected if Select=true
3     , typename T2 >    // Type to be selected if Select=false
4 struct SelectType
5 {
6     typedef T1  Type; // The selected type.
7 };
8
9 template< typename T1    // Type not to be selected
10     , typename T2 >    // Type to be selected
11 struct SelectType<false,T1,T2>
12 {
13     typedef T2  Type; // The selected type.
14 };

```

In case the first given template argument **Select** is **true**, the base template is selected, which has a nested type definition for the second template argument (the first of the two given types to select from). In case **Select** is **false**, the specialization is selected, which defines a nested type for the third template argument (the second of the two given types). Note that this class merely selects one of two given types. It is therefore not necessary for the compiler to instantiate any of the given types.

B The MathTrait class template

The **MathTrait** class template offers the possibility to select the resulting data type of a generic mathematical operation. Listing 41 demonstrates the use of **MathTrait** for the addition of two values. Via **MathTrait** it is possible to specify the return type of the **add** function without knowledge about the two given types **T1** and **T2**:

Listing 41: Application of the MathTrait class template

```

1 template< typename T1          // The type of the left operand
2     , typename T2 >          // The type of the right operand
3 typename MathTrait<T1,T2>::AddType // The resulting generic return type
4 add( T1 t1, T2 t2 )          //
5 {                             // The function 'add' returns the sum
6     return t1 + t2;          // of the two given values
7 }

```

Per default, the **MathTrait** template provides specializations for all built-in data types (except **void** and **bool**, which are both not considered numeric data types, but instead including **std::size_t** and **std::ptrdiff_t** for several compilers) and all mathematical classes of the *pe*. Specifying the resulting data type for a specific operation is done by specializing the **MathTrait** template for this particular type combination. In case a certain type combination is not defined in a **MathTrait** specialization, the base template is selected, which is left undefined and therefore stops the compilation process:

Listing 42: Undefined base template of the MathTrait class

```

1 template< typename T1, typename T2 >
2 struct MathTrait;

```

Each specialization of **MathTrait** defines the data types **HighType** that represents the high-order data type of the two given data types and **LowType** that represents the low-order data type. Additionally, each specialization defines the types **AddType**, **SubType**, **MultType**, and **DivType**, that represent the type of the resulting data type of the corresponding mathematical operation. The following example shows the specialization for operations between the double and the integer type:

Listing 43: Specialization of the MathTrait class template

```

1 template<>
2 struct MathTrait< double, int >
3 {
4     typedef double  HighType;
5     typedef int     LowType;
6     typedef double  AddType;
7     typedef double  SubType;
8     typedef double  MultType;
9     typedef double  DivType;
10 };

```

In case of operations between built-in data types, the `MathTrait` class defines the more significant data type as the resulting data type. For this selection, signed data types are given a higher significance. It is also possible to specialize the `MathTrait` template for additional user-defined data types, such as vectors and matrices. However, it is possible that a specific mathematical operation is invalid for the particular type combination. In this case, the `INVALID_NUMERICAL_TYPE` can be used to fill the missing type definition. The `INVALID_NUMERICAL_TYPE` represents the resulting data type of an invalid numerical operation. It is left undefined to stop the compilation process in case it is instantiated. The following example shows the specialization of the `MathTrait` template for `MatrixMxN` and `VectorN`. In this case, only the multiplication between the matrix and the vector is a valid numerical operation. Therefore for all other types the `INVALID_NUMERICAL_TYPE` is used.

Listing 44: Specialization of the `MathTrait` class template

```

1 template< typename T1, typename T2 >
2 struct MathTrait< MatrixMxN<T1>, VectorN<T2> >
3 {
4     // Invalid, no common high data type
5     typedef INVALID_NUMERICAL_TYPE           HighType;
6
7     // Invalid, no common low data type
8     typedef INVALID_NUMERICAL_TYPE           LowType;
9
10    // Invalid, cannot add a matrix and a vector
11    typedef INVALID_NUMERICAL_TYPE           AddType;
12
13    // Invalid, cannot subtract a vector from a matrix
14    typedef INVALID_NUMERICAL_TYPE           SubType;
15
16    // Multiplication between a matrix and a vector
17    typedef VectorN< typename MathTrait<T1,T2>::MultType > MultType;
18
19    // Invalid, cannot divide a matrix by a vector
20    typedef INVALID_NUMERICAL_TYPE           DivType;
21 };

```

Note the recursive instantiation of `MathTrait` for the definition of the data type resulting from a multiplication between a 3×3 matrix and a three-dimensional vector. Both classes are defined as templates to enable arbitrary data types as elements:

Listing 45: Declarations of the `Matrix` and `Vector` classes

```

1 template< typename T > class MatrixMxN;
2 template< typename T > class VectorN;

```

Therefore it is possible to combine matrices and vectors with different element types. In the simplest case, a matrix of `double` values might be multiplied with a vector of `int` values. In this case, the resulting data type would be a three-dimensional vector of `double` values:

Listing 46: Mixed-type matrix-vector multiplication

```

1 MatrixMxN<double> A;
2 VectorN<int>      v;
3
4 VectorN<double> x = A * v;

```

A more complex example might involve a matrix of matrices of `float` values and a vector of vectors of `long` values:

Listing 47: Mixed-type matrix-vector multiplication

```

1 MatrixMxN< MatrixMxN<float> > A;
2 VectorN< VectorN<long> >      v;
3
4 VectorN< VectorN<float> > x = A * v;

```

Still the operation is well defined and the correct return type of the multiplication can be evaluated by the recursive use of the `MathTrait` class template.

C Further Performance Results

This section contains further performance comparisons between the *pe* and Boost uBLAS. It shows a complete evaluation of all possible operations between dense and sparse vectors and matrices to demonstrate the suitability of the smart expression template approach. For the measurements, the following data types from the Boost uBLAS library were used: `vector` for dense vectors, `matrix` for dense matrices, `compressed_vector` for sparse vectors, and `compressed_matrix` for sparse matrices. The according data types of the *pe* are `VectorN` for dense vectors, `MatrixMxN` for dense matrices, `SparseVectorN` for sparse vectors, and `SparseMatrixMxN` for sparse matrices. The test machine for all performance tests was an Intel Core i7 940 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache. All executables are compiled with the GNU G++ 4.4.1 compiler (branch revision 150839).

A general observation of the performance results is that the *pe* always performs better than the Boost uBLAS library (except for the dense matrix-dense matrix multiplication with small matrices). For all operations between dense data structures, the performance difference between *pe* and uBLAS is either very similar or within reasonable bounds. However, all operations involving a sparse data structure exhibit a tremendous performance advantage for the *pe*. The reason for this is based in the general expression template formulation of uBLAS as well as in the optimization efforts of the *pe*.

C.1 Dense Vector-Dense Vector Addition/Subtraction

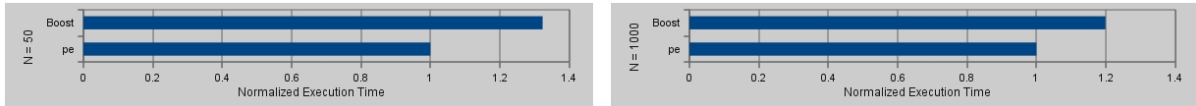


Figure 9: Performance comparison between the *pe* and Boost for the dense vector dense vector addition/subtraction.

C.2 Dense Vector-Sparse Vector Addition/Subtraction

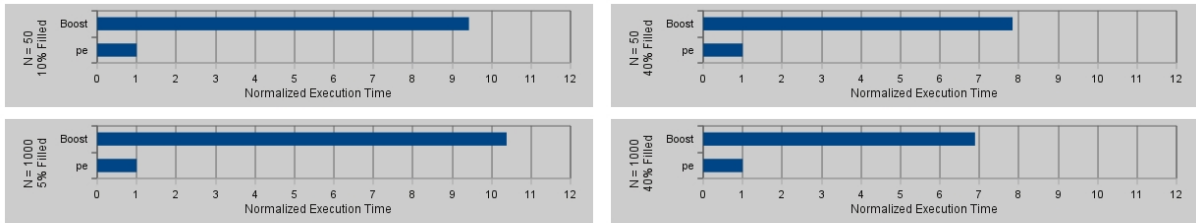


Figure 10: Performance comparison between the *pe* and Boost for the dense vector sparse vector addition/subtraction.

C.3 Sparse Vector-Dense Vector Addition/Subtraction

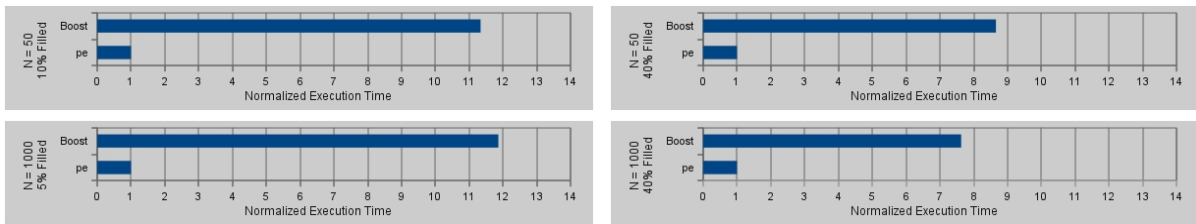


Figure 11: Performance comparison between the *pe* and Boost for the sparse vector dense vector addition/subtraction.

C.4 Sparse Vector-Sparse Vector Addition/Subtraction

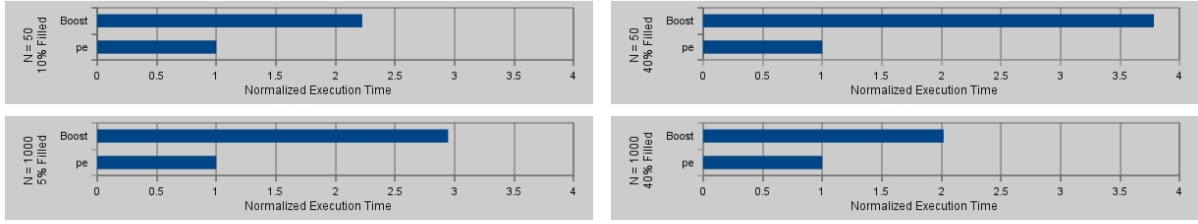


Figure 12: Performance comparison between the *pe* and Boost for the sparse vector sparse vector addition/subtraction.

C.5 Dense Vector-Scalar Multiplication

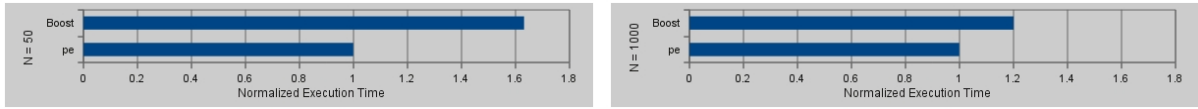


Figure 13: Performance comparison between the *pe* and Boost for the multiplication of a dense vector and a scalar.

C.6 Sparse Vector-Scalar Multiplication

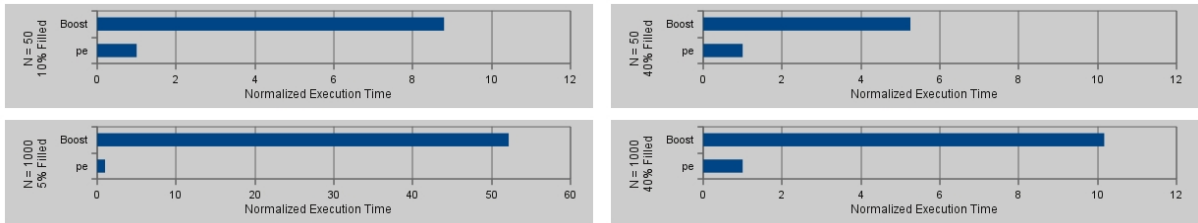


Figure 14: Performance comparison between the *pe* and Boost for the multiplication of a sparse vector and a scalar.

C.7 Dense Matrix-Dense Vector Multiplication

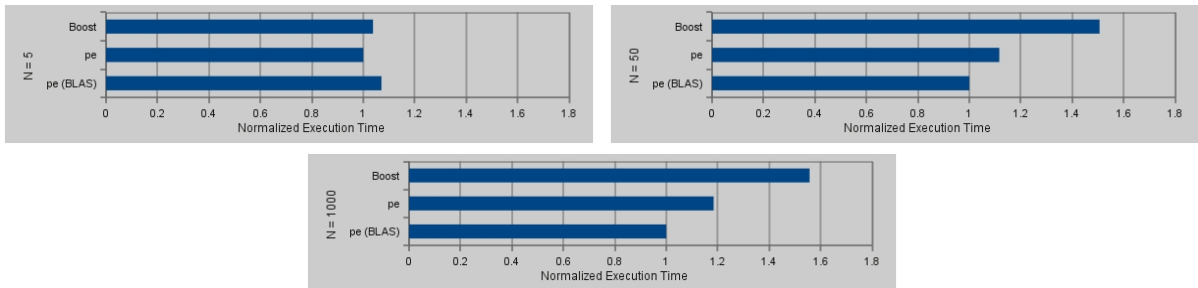


Figure 15: Performance comparison between the *pe* and Boost for the dense matrix dense vector multiplication.

C.8 Dense Matrix-Sparse Vector Multiplication



Figure 16: Performance comparison between the *pe* and Boost for the dense matrix sparse vector multiplication.

C.9 Sparse Matrix-Dense Vector Multiplication

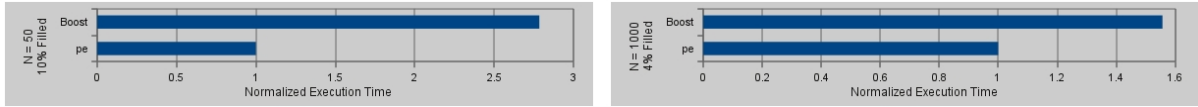


Figure 17: Performance comparison between the *pe* and Boost for the sparse matrix dense vector multiplication.

C.10 Sparse Matrix-Sparse Vector Multiplication

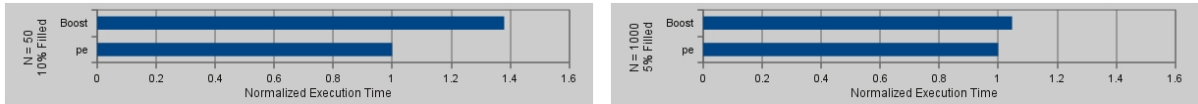


Figure 18: Performance comparison between the *pe* and Boost for the sparse matrix sparse vector multiplication.

C.11 Dense Matrix-Dense Matrix Addition/Subtraction



Figure 19: Performance comparison between the *pe* and Boost for the dense matrix dense matrix addition/subtraction.

C.12 Dense Matrix-Sparse Matrix Addition/Subtraction

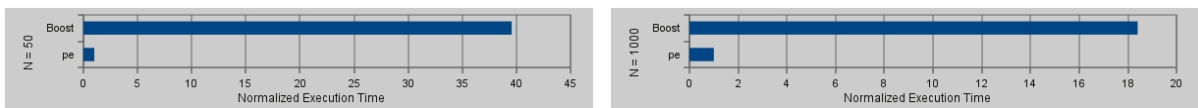


Figure 20: Performance comparison between the *pe* and Boost for the dense matrix sparse matrix addition/subtraction.

C.13 Sparse Matrix-Dense Matrix Addition/Subtraction

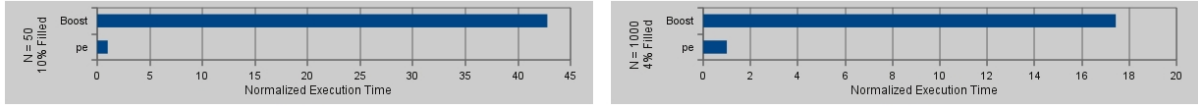


Figure 21: Performance comparison between the *pe* and Boost for the sparse matrix dense matrix addition/subtraction.

C.14 Sparse Matrix-Sparse Matrix Addition/Subtraction

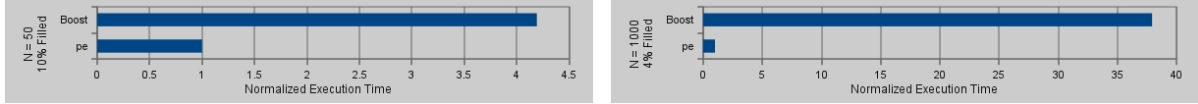


Figure 22: Performance comparison between the *pe* and Boost for the sparse matrix sparse matrix addition/subtraction.

C.15 Dense Matrix-Scalar Multiplication

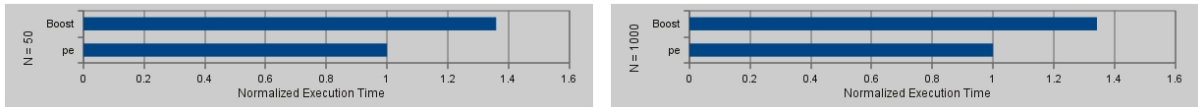


Figure 23: Performance comparison between the *pe* and Boost for the multiplication of a dense matrix and a scalar.

C.16 Sparse Matrix-Scalar Multiplication

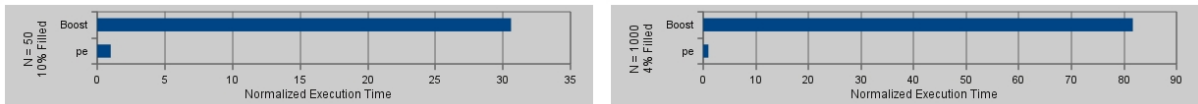


Figure 24: Performance comparison between the *pe* and Boost for the multiplication of a sparse matrix and a scalar.

C.17 Dense Matrix-Dense Matrix Multiplication

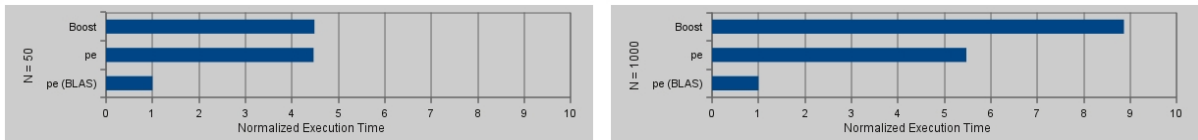


Figure 25: Performance comparison between the *pe* and Boost for the dense matrix dense matrix multiplication.

C.18 Dense Matrix-Sparse Matrix Multiplication

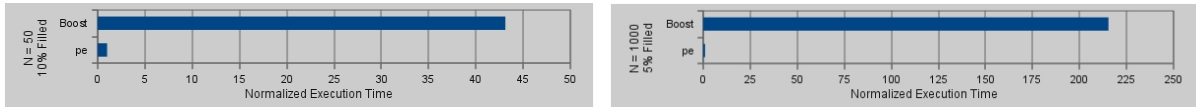


Figure 26: Performance comparison between the *pe* and Boost for the dense matrix sparse matrix multiplication.

C.19 Sparse Matrix-Dense Matrix Multiplication



Figure 27: Performance comparison between the *pe* and Boost for the sparse matrix dense matrix multiplication.

C.20 Sparse Matrix-Sparse Matrix Multiplication

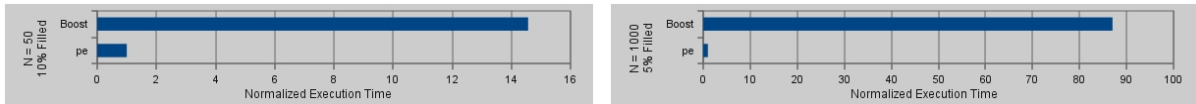


Figure 28: Performance comparison between the *pe* and Boost for the sparse matrix sparse matrix multiplication.

References

- [1] AMD Core Math Library (ACML). Homepage of the ACML framework: <http://www.amd.com/acml>.
- [2] Automatically Tuned Linear Algebra Software (ATLAS). Homepage of the ATLAS framework: <http://math-atlas.sourceforge.net/>.
- [3] Boost. Homepage of the Boost C++ framework: <http://www.boost.org>.
- [4] Boost uBLAS library. Homepage of the Boost uBLAS library: http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/index.htm.
- [5] S.C. Dewhurst. *C++ Gotchas*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2007.
- [6] H. Sutter. *More Exceptional C++*. C++ In-Depth Series. Addison/Wesley, 2007.
- [7] H. Sutter. *Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2008.
- [8] J. Härdtlein. *Moderne Expression Templates Programmierung - Weiterentwickelte Techniken und deren Einsatz zur Lösung partieller Differentialgleichungen*. PhD thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2007.
- [9] K. Iglberger and U. Rüde. C++ Compile Time Constraints. Technical Report 09-10, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2009.
- [10] Intel Math Kernel Library (IMKL). Homepage of the IMKL framework: <http://www.intel.com/software/products/mkl>.
- [11] K. Iglberger and U. Rüde. The *pe* Rigid Multi-Body Physics Engine. Technical Report 09-9, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2009.

- [12] S.B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 10th printing edition, 2007.
- [13] NVidia CUDA BLAS Library. Homepage of the CUBLAS framework: <http://www.nvidia.com/>.
- [14] T. Veldhuizen. Expression templates. In *C++ Report*, volume 7, pages 26–31. 1995.
- [15] T. Veldhuizen. Expression templates. In *C++ Gems*, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.
- [16] D. Vandevorde and N.M. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2003.
- [17] M. Wilson. *Imperfect C++*. Addison-Wesley, 2005.